I'll analyze both services and explain every function in detail. Let me break down each service systematically.

**GradeService (src/services/grade\_service.py)**

The GradeService is the core orchestrator for AI-powered grading in the system. Here's every function:

**Helper Functions & Classes**

**\_row\_to\_ns(row: Any) -> SimpleNamespace**

* **Purpose**: Converts SQLAlchemy database rows to attribute-accessible namespace objects
* **Usage**: Makes database results easier to work with by allowing dot notation access

**GradingMetrics (Dataclass)**

* **Purpose**: Tracks performance metrics during grading
* **Fields**: Processing time, concept extraction time, semantic analysis time, rubric application time, total LLM calls, confidence score

**GradingError(Exception)**

* **Purpose**: Custom exception for grading-related errors
* **Usage**: Provides specific error handling for grading failures

**SemanticAnalyzer Class**

**\_\_init\_\_(self)**

* **Purpose**: Initializes the semantic analyzer with LLM service
* **Dependencies**: Uses the global llm\_service instance

**extract\_key\_concepts(ideal\_answer: IdealAnswer) -> List[KeyConcept]**

* **Purpose**: Extracts key concepts from an ideal answer using AI
* **Process**:

1. Calls LLM service to analyze the ideal answer
2. Extracts concepts with importance, keywords, and explanations
3. Converts to KeyConcept objects

* **Returns**: List of key concepts with metadata
* **Error Handling**: Raises GradingError on failure

**analyze\_semantic\_similarity(ideal\_answer, student\_answer, key\_concepts) -> Dict[str, Any]**

* **Purpose**: Analyzes how similar student answer is to ideal answer
* **Process**:

1. Converts key concepts to dict format
2. Calls LLM service for semantic analysis
3. Returns detailed analysis results

* **Returns**: Dictionary with similarity scores and concept evaluations
* **Error Handling**: Raises GradingError on failure

**ResponseEvaluator Class**

**\_\_init\_\_(self)**

* **Purpose**: Initializes the response evaluator with LLM service
* **Dependencies**: Uses the global llm\_service instance

**apply\_rubric(ideal\_answer, student\_answer, semantic\_analysis) -> Dict[str, Any]**

* **Purpose**: Applies grading rubric to calculate scores and feedback
* **Process**:

1. Converts rubric to dict format
2. Calls LLM service with rubric data and semantic analysis
3. Returns scoring results

* **Returns**: Dictionary with scores, feedback, and criteria breakdown
* **Error Handling**: Raises GradingError on failure

**chain\_of\_thought\_grading(ideal\_answer, student\_answer) -> Dict[str, Any]**

* **Purpose**: Performs comprehensive Chain-of-Thought grading (recommended approach)
* **Process**:

1. Converts rubric to dict format
2. Calls LLM service for comprehensive analysis
3. Returns detailed step-by-step grading results

* **Returns**: Dictionary with complete grading analysis
* **Error Handling**: Raises GradingError on failure

**GradeService Main Class**

**\_\_init\_\_(db\_manager: DatabaseManager = None)**

* **Purpose**: Initializes the grading service
* **Components**: Creates semantic analyzer, response evaluator, and LLM service instances
* **Database**: Optional database manager for persistence

**get\_session() -> Session**

* **Purpose**: Gets database session for data operations
* **Error Handling**: Raises RuntimeError if database manager not available
* **Usage**: Used for database operations in grading workflow

**grade\_answer(student\_answer, ideal\_answer, use\_chain\_of\_thought=True) -> GradingResult**

* **Purpose**: Main grading function that orchestrates the entire grading process
* **Parameters**:
* student\_answer: Student's submitted answer
* ideal\_answer: Reference answer with rubric
* use\_chain\_of\_thought: Whether to use CoT approach (recommended)
* **Process**:

1. Tracks performance metrics
2. Chooses grading approach (CoT or step-by-step)
3. Returns complete grading result

* **Returns**: GradingResult with scores, feedback, and metadata
* **Error Handling**: Raises GradingError on failure

**\_grade\_with\_chain\_of\_thought(student\_answer, ideal\_answer, metrics) -> GradingResult**

* **Purpose**: Implements Chain-of-Thought grading approach (recommended)
* **Process**:

1. Calls response evaluator for comprehensive analysis
2. Extracts results from CoT response
3. Converts concept comparisons to ConceptEvaluation objects
4. Calculates final scores and pass status
5. Creates complete GradingResult

* **Returns**: Complete grading result with all metrics
* **Features**: More comprehensive analysis, better accuracy

**\_grade\_step\_by\_step(student\_answer, ideal\_answer, metrics) -> GradingResult**

* **Purpose**: Implements step-by-step grading approach (alternative)
* **Process**:

1. Extracts key concepts if not present
2. Analyzes semantic similarity
3. Applies grading rubric
4. Constructs final result

* **Returns**: Complete grading result
* **Features**: More granular control, step-by-step analysis

**\_extract\_similarity\_score(cot\_result: Dict[str, Any]) -> float**

* **Purpose**: Extracts semantic similarity score from Chain-of-Thought result
* **Process**: Calculates weighted average based on concept accuracy
* **Returns**: Similarity score between 0 and 1
* **Fallback**: Returns 0.8 if no concept comparisons available

**\_calculate\_completeness\_score(concept\_evaluations: List[ConceptEvaluation]) -> float**

* **Purpose**: Calculates completeness score based on concept coverage
* **Process**: Counts present concepts vs total concepts
* **Returns**: Completeness score between 0 and 1
* **Fallback**: Returns 0.7 if no concept evaluations available

**batch\_grade(request: BatchGradingRequest) -> BatchGradingResponse**

* **Purpose**: Grades multiple answers in batch for efficiency
* **Process**:

1. Iterates through all grading requests
2. Grades each answer individually
3. Tracks success/failure counts
4. Returns batch response with all results

* **Returns**: BatchGradingResponse with all results and statistics
* **Error Handling**: Individual failures don't stop batch processing

**complete\_grading\_workflow(question\_id: int, student\_id: int) -> Dict[str, Any]**

* **Purpose**: Complete end-to-end grading workflow
* **Process**:

1. Retrieves ideal answer and marks
2. Extracts and saves key concepts
3. Retrieves student's submitted answer
4. Grades and saves results

* **Returns**: Grading result in required format
* **Dependencies**: Uses RAG service for data retrieval
* **Error Handling**: Raises ValueError for missing data

**LLMService (src/services/llm\_service.py)**

The LLMService handles all interactions with Language Learning Models. Here's every function:

**Exception Classes**

**LLMError(Exception)**

* **Purpose**: Base exception for LLM-related errors
* **Usage**: General LLM operation failures

**LLMProviderError(LLMError)**

* **Purpose**: Provider-specific errors
* **Usage**: API connection, authentication, or provider-specific issues

**LLMResponseParsingError(LLMError)**

* **Purpose**: Response parsing errors
* **Usage**: When LLM returns invalid JSON or malformed responses

**BaseLLMProvider (Abstract Base Class)**

**\_\_init\_\_(api\_key: str, model: str)**

* **Purpose**: Base initialization for all LLM providers
* **Parameters**: API key and model name
* **Config**: Loads model-specific configuration

**generate\_response(prompt, temperature, max\_tokens, json\_mode) -> str (Abstract)**

* **Purpose**: Abstract method for generating responses
* **Parameters**: Prompt, temperature, max tokens, JSON mode flag
* **Returns**: Generated response string

**validate\_connection() -> bool (Abstract)**

* **Purpose**: Abstract method for connection validation
* **Returns**: Boolean indicating connection status

**GitHubModelsProvider Class**

**\_\_init\_\_(github\_token: str, model: str, endpoint: str)**

* **Purpose**: Initializes GitHub Models provider
* **Parameters**: GitHub token, model name, custom endpoint
* **Client**: Creates OpenAI client with custom base URL

**generate\_response(prompt, temperature, max\_tokens, json\_mode) -> str**

* **Purpose**: Generates response from GitHub Models API
* **Features**:
* Retry mechanism with exponential backoff
* JSON mode support
* Temperature and token limits
* **Error Handling**: Converts OpenAI errors to provider errors
* **Returns**: Generated response content

**validate\_connection() -> bool**

* **Purpose**: Validates GitHub Models connection
* **Process**: Sends simple test request
* **Returns**: Boolean indicating connection status
* **Error Handling**: Returns False on any connection failure

**LLMService Main Class**

**\_\_init\_\_(self)**

* **Purpose**: Initializes the LLM service
* **Process**: Sets up provider based on configuration
* **Dependencies**: Uses global settings for configuration

**initialize\_provider() -> None**

* **Purpose**: Initializes the appropriate LLM provider
* **Process**:

1. Checks provider type from settings
2. Creates provider instance
3. Validates configuration

* **Error Handling**: Raises LLMError for unsupported providers or missing tokens

**extract\_key\_concepts(ideal\_answer: str, subject: str, topic: str) -> List[Dict[str, Any]]**

* **Purpose**: Extracts key concepts from ideal answers using AI
* **Process**:

1. Formats prompt with ideal answer, subject, and topic
2. Calls LLM with concept extraction prompt
3. Parses JSON response

* **Returns**: List of key concepts with metadata
* **Error Handling**: Raises LLMError on failure

**analyze\_semantic\_similarity(ideal\_answer, student\_answer, key\_concepts) -> Dict[str, Any]**

* **Purpose**: Analyzes semantic similarity between answers
* **Process**:

1. Formats key concepts as JSON string
2. Calls LLM with semantic analysis prompt
3. Parses JSON response

* **Returns**: Dictionary with similarity analysis
* **Error Handling**: Raises LLMError on failure

**apply\_grading\_rubric(ideal\_answer, student\_answer, rubric, concept\_evaluations, semantic\_analysis) -> Dict[str, Any]**

* **Purpose**: Applies grading rubric to calculate scores
* **Process**:

1. Formats rubric and concept evaluations as JSON
2. Calls LLM with rubric application prompt
3. Parses JSON response

* **Returns**: Dictionary with scoring results
* **Error Handling**: Raises LLMError on failure

**chain\_of\_thought\_grading(ideal\_answer, student\_answer, subject, rubric) -> Dict[str, Any]**

* **Purpose**: Performs comprehensive Chain-of-Thought grading
* **Process**:

1. Formats rubric as JSON string
2. Calls LLM with CoT grading prompt
3. Parses JSON response

* **Returns**: Dictionary with complete grading analysis
* **Error Handling**: Raises LLMError on failure

**\_parse\_json\_response(response: str) -> Dict[str, Any]**

* **Purpose**: Parses JSON response with error handling
* **Process**:

1. Strips markdown formatting if present
2. Parses JSON content
3. Handles parsing errors gracefully

* **Returns**: Parsed JSON dictionary
* **Error Handling**: Raises LLMResponseParsingError on JSON parse failure

**validate\_connection() -> bool**

* **Purpose**: Validates LLM service connection
* **Process**: Delegates to provider's validation method
* **Returns**: Boolean indicating connection status

**get\_provider\_info() -> Dict[str, Any]**

* **Purpose**: Gets information about current provider and model
* **Returns**: Dictionary with provider details, model info, configuration, and connection status
* **Features**: Includes provider-specific information (GitHub endpoint, token status)

**Global Instance**

**llm\_service = LLMService()**

* **Purpose**: Global instance of LLM service
* **Usage**: Imported and used throughout the application
* **Initialization**: Automatically initializes on import

**Key Relationships**

1. **GradeService** orchestrates the entire grading process
2. **LLMService** provides AI capabilities for analysis
3. **SemanticAnalyzer** uses LLM service for concept extraction and similarity analysis
4. **ResponseEvaluator** uses LLM service for rubric application and CoT grading
5. Both services work together to provide comprehensive AI-powered grading functionality

The system is designed with proper error handling, retry mechanisms, and modular architecture for maintainability and reliability.